REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
~~31 March 1998~~

3. REPORT TYPE AND DATES COVERED
Technical Report

4. TITLE AND SUBTITLE
Automation Comparison Procedure for Verification of Hybrid Sistems

5. FUNDING NUMBERS
DAAH04-96-1-0341

6. AUTHOR(S)
W. Kohn, J.B. Remmel and A. Nerode

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Regents of the University of California
c/o Sponsored Projects Office
336 Sproul Hall
Berkeley, CA 94720-5940

8. PERFORMING ORGANIZATION
REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U. S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING
AGENCY REPORT NUMBER
ARO 35873.79-MA-MUR

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by the documentation.

12 a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This paper describes on-going research on a procedure for the verification of hybrid system controllers implemented via HyBrithms' Multiple Agent Hybrid Control Architecture which executes the Kohn-Nerode procedure for the on-line extraction of real-time controls. It is an automated static verification technique based on the construction of an Intersection Unification Automaton We discuss an essential step of this verification technique, namely, a procedure to verify that the controller design generated by an agent in MAHCA meets the requirements established for that agent. The paper describes the functionality of the procedure and illustrates it with an example

14. SUBJECT TERMS
verification, hybrid system controllers, control automaton

15. NUMBER OF PAGES
6

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION ON THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Center for
Foundations of
Intelligent
Systems

Technical Report
97-11

Automaton Comparison Procedure for the Verification of Hybrid Systems

W. Kohn, J. B. Remmel and A. Nerode

November 1997

CORNELL
UNIVERSITY

625 Rhodes Hall, Ithaca, NY 14853 (607) 255-8005

Technical Report
97-11

Automaton Comparison Procedure for the Verification of Hybrid Systems

W. Kohn, J. B. Remmel and A. Nerode

November 1997

Automaton Comparison Procedure for the Verification of Hybrid Systems

Wolf Kohn[*]**and Jeffrey B. Remmel**[†]
HyBrithms Corporation[‡] 11201 SE 8th Street #J140
Bellevue, WA 98004-6420
e-mail: wk@hybrithms.com, jremmel@hybrithms.com
Anil Nerode[§]
Mathematical Sciences Institute, Cornell University
Ithaca, NY 14853
e-mail: anil@math.cornell.edu

Abstract

This paper describes on-going research on a procedure for the verification of hybrid system controllers implemented via HyBrithms' Mutiple Agent Hybrid Control Architecture which executes the Kohn-Nerode procedure for the on-line extraction of real-time controls. It is an automated static verification technique based on the construction of an Intersection Unification Automaton (IUA). We discuss an essential step of this verification technique, namely, a procedure to verify that the controller design generated by an agent in MAHCA meets the requirements established for that agent. The paper descibes the functionality of the procedure and illustrates it with an example.

1. Introduction

This paper describes on-going research on a procedure for the verification of hybrid system controllers implemented via HyBrithms' Multiple Agent Hybrid Control Architecture [11] which executes the Kohn-Nerode procedure, [8, 10, 12, 13, 14, 15], for the on-line extraction of real-time control. It is an automated static verification technique based on the construction of an Intersection Unification Automaton (IUA). Given a hybrid system controller generated by a MAHCA agent and a set of control specifications for its desired behavior, the proposed verification procedure builds an IUA constructed by an automata operation on two inference automata: the automaton encoding the specifications and the control automaton constructed by the inferencer of the agent [11]. The ultimate goal of our research is to develop a verification procedure to verify a MAHCA distributed implementation to a set of global requirement specifications. In this paper we discuss an essential step, namely, a procedure to verify that the controller design generated by an agent in MAHCA meets the requirements established for that agent. The paper descibes the functionality of the procedure and illustrates it with an example.

2. The outline of the verification procedure

The verification procedure that we propose is based on the construction of a proof system in the domain in which a hybrid controller generated by an agent's inferencer and the specifications are represented by nondetermistic finite state automata.

The proof system is very simple. It is based on the comparison of the behavior of the two automata in the domain: one representing the execution automaton of an agent's inference automaton [11] (in the domain of the proof system) and the other representing the specifications. In the remaining part of this section we shall give an outline of our proof procedure.

Consider a control automaton with requirements. Let A be the behavior associated with the agent and let B be the behavior of the automaton representing the requirements. The proof system has to show that for any inputs to A , the behavior R , defined by

$$R = (A \cap_u (B)^c)^t \tag{1}$$

is empty where $\cap_u$ is a binary operation on the universe of behaviors such that if $Z = X \cap_u Y$ for behaviors X and Y in the universe of behaviors U, then Z is the behavior common to X and Y. In (1), $()^c$ is a unary operation on the universe of behaviors such that if $Z = (X)^c$, then Z is the logical complement of the behavior X . Intuitively, if X is a behavior representing a set of requirements, $(X)^c$

[*] Research supported by SDIO contract DAA H-04-93-C-0113, Dept. of Commerce Agreement 70-NANB5H1164.

[†] Research supported by SDIO contract DAA H-04-93-C-0113, Dept. of Commerce Agreement 70-NANB5H1164. On leave from the Department of Mathematics, University of California at San Diego.

[‡] HyBrithms Corporation was formerly known as Sagent Corporation

[§] Research Supported by ARO under the MURI program "Integrated Approach to Intelligent Systems," grant no. DAAHO4-96-1-0341.

represents the set of behaviors which violate those requirements. Finally in (1), $()^t$ is a unary operation on U such that if $Z = (X)^t$, then Z represent the smallest behavior equivalent to X.

3. Verification automaton

A verification automaton is an object
$A = (Q, X, I, T, Y, Z, \Phi, \delta, \alpha, \beta)$ where
1) Q is a set of states.
2) X is the domain set.
3) I is the set of initial states.
4) T is the set of terminal states.
5) Y is the input domain.
6) Z is the output domain.
7) $\Phi = \{\phi_i | \phi_i : X \to X\}$ is the instruction set.
8) $\delta : Q \times \Phi \to Q$ is the state transition function.
9) $\alpha : Y \to X$ is the input function.
10) $\beta : Q \times \Phi \times X \to Z$ is the output function.

In addition, we make the following assumptions about the verification automata.

I. We ignore the sensor and goal domains which are the inputs to the control automaton A and the output domain which is the set of control actions.

II. Each element $q \in Q$ is a controller state. The controller states corresponds to relations which the system must satisfy after the excution of the control action which is issued in the controller state, see [5].

III. The relation δ encodes the transition between relations to be satisfied. If q is the current controller state which implements action $\phi_i \in \Phi$ and $x \in X$ is the current state of the system, then $\delta(q, \phi_i(x)) = q_{next}$ where q_{next} is the label of the next relation to be satisfied.

IV. If $q \in Q$ is an element of I, we say that q is an initial relation label.

V. If $q \in Q$ is an element of T, we say that q is a success relation label.

VI. The output function β satisfies the following:

$$\beta(q, \phi_i(x)) = \begin{cases} \phi_i(x) & \text{if } q \in T, \\ \perp & \text{if otherwise} \end{cases} \tag{2}$$

That is, if q is the label of the control $\phi_i(x)$ and q is terminal, then the output function gives the value of the control action. Otherwise the output function generates the empty symbol $\perp$.

An iterated state transition represents a sequence of two or more consecutive state transitions coresponding to a control action satisfying the composite of the corresponding relations. Let Φ^* be the monoid of all control actions under chattering composition [5]

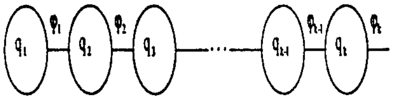

Figure 1:

$$\Phi^* = \cup_{k=0}^{\infty} \Phi^k. \tag{3}$$

Here $\Phi^0 = \{\wedge\}$ with $\wedge$ being the identity function on X so that $\wedge(x) = x$ for all $x \in X$ and Φ^k is the set of all chatering compositions with exactly k steps

$$\phi_{i_k}(\phi_{i_{k-1}} \cdots (\phi_{i_2}(\phi_{i_1}())) \cdots).$$

This corresponds for example to a branch of the controller whose state path in the automaton is pictured in Figure 1.

The iterated state transition δ^* is a relation $\delta^* : Q \times \Phi^* \to Q$, defined recursively as follows:

$$\begin{array}{rcll} \delta^*(q, \wedge) & = & q & \\ \delta^*(q, \phi) & = & \phi & \forall \phi \in \Phi \\ \delta^*(q, \phi\omega) & = & \delta^*(\delta(q, \phi), \omega) & \forall \phi \in \Phi, \omega \in \Phi^* \\ & \text{or} & & \\ \delta^*(q, \omega\phi) & = & \delta(\delta^*(q, \omega), \phi) & \forall \phi \in \Phi, \omega \in \Phi^* \end{array}$$

The state behavior of the automaton defines a function $A_q : \Phi^* \times X \to X$ each $q \in Q$ by

$$A_q(\omega, x) = \begin{cases} \omega(x) & \text{if } \delta^*(q, \omega) \in T \\ \perp & \text{if otherwise.} \end{cases} \tag{4}$$

Thus any control action applied to an x in the domain X yields a result that is still in the domain.

The output behavior function is denoted by OA_q and maps the $Y \times \Phi^*$ where Y is the input set to the output set Z by

$$OA_q(y) = \beta(A_q(\omega, \alpha(y))). \tag{5}$$

Given an automaton $A = (Q, X, I, T, Z, \Phi, \delta, \alpha, \beta)$, the *accessible automaton* A^a associated with A is defined by the following algorithm in which $Q_0, Q_1, \ldots$ represent sets of states, q' represents a new state, q represents a previous state, $Q_0 = I$, $Q_{i+1} = \{q' \in Q - Q_i | \exists q \in Q_i \exists \phi \in \Phi(\delta(q, \phi) = q')\}$, and with the termination criterion that if $Q_k = \emptyset$, then $Q_{k+p} = \emptyset$ for all $p \geq 0$.

This algorithm defines a sequence of pairwise disjoints sets of states $Q_0, Q_1, \ldots$ such that each state set Q_{i+1} represents all states q' that can only be reached from previous state $q \in Q_i$. Any states that cannot be reached from a previous state are considered dead code. The algorithm terminates when no new states are defined from previous states.

Define the *accessible state set* Q^a to be set of states that can be reached from some initial state $q \in I$ and define the *accessable terminal state set* (or the set of terminal state accessible from I) T^a by i $T^a = T \cap Q^a$. We then define the accessible automaton A^a by $A^a = (Q^a, X, T^a, I^a, Y, Z, \Phi, \delta^a, \alpha, \beta)$ where $I^a = Q^a \cap I = I$ and δ^a is the restriction of δ to $Q^a \times \Phi$. Note that if $|Q| = n$, then since the sets Q_i are pairwise disjoint we have $Q_n = \emptyset$. Thus we have an a priori bound on the length of the procedure.

Given the automaton A, a coaccessible automaton can be built in a similar manner. To do this, we need to define the reversal automaton A^r. The idea is simple. We want a state path of the reversal automaton to the state path of A except that we want to traverse the state path backwards. Thus if $A = (Q, X, I, T, Y, Z, \Phi, \delta, \alpha, \beta)$, then $A^r = (Q, X, I^r, T^r, Y, Z, \Phi, \delta^r, \alpha, \beta)$ where $I^r = T$, $T^r = I$, and $\delta^r(q, \phi) = q'$ if $\delta(q', \phi) = q \; \forall q, q' \in Q \forall \phi \in \Phi$ An automaton state $q \in Q$ is said to be *coaccessible* if and only if its q is is accessible in the reversal automaton. We then define the *coaccessible automaton* A^c by

$$A^c = ((A^r)^a)^r.$$

That is, the coaccessible automaton A^c is obtained by reversing automaton A, computing the accessible automaton of the reversal automaton , and then reversing the resulting automaton. Finally, given an automaton A, the trim automaton A^t associated with A is defined

$$A^t = (A^a)^c = (A^c)^a.$$

That is, to compute the trim automaton A^t, we compute the accessible automaton A^a and compute for it the coaccessible automaton of A^a. The most important property of a trim automaton is that every state path segment is a path from an initial to a terminal state. Computing the trim automaton gets rid of dead code automatically (this includes code that is unnecessary, which doesn't contribute to the output).

4. Intersection Unification Automaton

Given automata

$$\begin{aligned} A_1 &= (Q_1, X, I_1, T_1, Y, Z, \Phi_1, \delta_1 \alpha_1, \beta_1) \text{ and} \qquad (6) \\ A_2 &= (Q_2, X, I_2, T_2, Y, Z, \Phi_2, \delta_2, \alpha_2, \beta_2), \qquad (7) \end{aligned}$$

the intersection unification automaton, denoted by $A_1 \cap A_2$ is defined as follows.

$$A_1 \cap A_2 = (Q_1 \times Q_2, X, I_1 \times I_2, T_1 \times T_2, Y, Z, \Phi_1, \delta_1 \alpha_1, \beta_1)$$

where $\forall q_1 \in Q_1, \forall q_2 \in Q_2, \forall \phi_1 \in \Phi_1, \forall \phi_2 \in \Phi_2$,

$$\delta((q_1, q_2), \phi) = \begin{cases} (\delta(q_1, \phi_1)), (\delta_2(q_2, \phi_2)) & \text{if (*)} \\ \perp & \text{if otherwise} \end{cases}$$

where (*) stands for "$\delta(q_1, \phi_1), (\delta_2(q_2, \phi_2)$ are defined and ϕ_1 unifies with ϕ_2".

The intersection of two automata is performed by taking the cross product of the states (Q), the initial states (I), and the terminal states (T) (the cross product generates all possible pairs of states between the two automatons) and keeping only the state transitions in the intersection automaton where the state transitions in the two original automatons are defined and the two state transitions unify. Two instructions unify if one can be used to implement the other. For example, an if statement can be implemented with a while-do loop.

In summary, the verification is performed by representing two different descriptions of a program, such as the program code and its requirements, as automatons and then intersecting the two automatons. In the intersection process, all possible pairs of states in the two automatons are examined and the state pairs that fail to unify are discarded. All states that are not members of a unified state pair are identified. Leftover states from the program automaton represent code that exceeds the requirements. Leftover states from the requirements automaton represent requirements that were not implemented.

5. The INVERT Prototype

The first phase of the tool, the translated-code verification tool, has been prototyped for a subset of the XPL language. For this phase of the tool, verification is defined as showing that the translated software meets the requirements established by the source language software; that is, showing that the two programs functionally perform the same and their output data undergo the same transitions. Since we wanted to concentrate on demonstrating the unification technique instead of language parsing techniques, the prototype was developed for only one language instead of two. Therefore, two different programs written in the same language are used to simulate the translated code.

This section contains a functional description of the prototype translated-code verification tool. It is described in terms of its software components and their functionality. The next section describes a sample execution of the prototype tool. The prototype tool was developed on an Sagent IRAD project by Dan Strauss, Robert St. John, and Holly Gibbons using the verification theory provided by Dr. Wolf Kohn.

Figure 3 illustrates the software components of INVERT and how they interact. INVERT consists of a Parser, Semantic Generator, Trimmer, and Unifier. The Parser and Semantic Generator together are abstractly referred to as the Automaton Generator. The Trimmer and Unifier together are abstractly referred to as the Automaton Manipulator. In the prototype INVERT, the Unifier is executed separately from the other components.

All components of the prototype INVERT were originally coded in the C Language Integrated Production System (CLIPS), an expert system tool developed by the Software Technology Branch at NASA Johnson Space Center

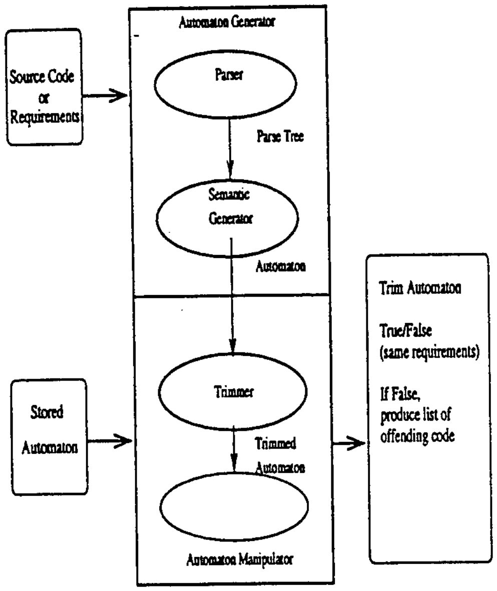

Figure 2: INVERT Software Components.

(JSC); however, some of the components have been converted to the C language.

The Parser accepts an XPL program and produces a parse tree based on the syntax (the grammar or structure) of the XPL language. As stated above, the INVERT prototype is implemented for a subset of the XPL language. The subset includes declarations, assignments, if-then-else statements, loops, bit variables, arrays, logical expressions, and arithmetic expressions. It does not include procedures, case statements, goto statements, and input/output statements. Therefore, the programs accepted by the prototype tool are simple, procedural, non-real-time XPL programs with no input or output statements. The Parser was developed using lex and yacc, the UNIX lexical analyzer and parser, respectively, and is based on the Backus-Naur form (BNF) of the XPL language.

The Semantic Generator takes the parse tree generated by the Parser and uses the operational semantic rules of XPL to produce a locally finite automaton describing all of the program's states for a given set of inputs. Physically, the automaton is a collection of data structures, or objects, that describe the order of the program instructions executed and the data contents changed after each instruction is executed. Specifically, a separate object describes each data item, each instruction, and each program state. The automaton serves as input to the Trimmer and the Unifier. The operational semantic rules of XPL, which define the meaning of the XPL statements, were obtained from the XPL language specification; test cases were used to determine the semantics when the specification lacked sufficient detail.

The Trimmer accepts the automaton and identifies in it the unnecessary code. Note that eliminating unnecessary code does not change the functionality of the program. The code must be trimmed to maximize the efficiency of the verification – only necessary code should be verified. The Trimmer in the prototype INVERT prompts the user for a list of output variables for the program, then traverses the program backwards and builds a list of variables that contribute to the output. Initially, only the output variables are on the list. As the traversal continues, the list grows to include all important intermediate variables. If a program statement does not alter any variables on the list, then it is considered unnecessary and is flagged as such in the automaton. The prototype Trimmer does not yet flag dead code because dead code is identifiable in the automaton generated by the Semantic Generator, which marks each statement as it is executed.

The Unifier takes two automatons produced by separate runs of the Automaton Generator and a list of desired output variables for one of the programs and, by comparing the states of the two automatons, tells whether they unify. Unification is defined as the output data undergoing the same transitions in both automatons. The Unifier uses variable substitution to map corresponding variables in the two automatons, so the two programs can use different names for the same variables. The Unifier then generates a Cartesian product of the output states from the first automaton with the output states from the second automaton, and examines every pair of output states to locate the pairs that unify (output states correspond to statements that modify output variables). Two output states unify if their variables substitute, they are assigned the same value, and the variables they reference have the same value. Two automatons unify if and only if all output states unify. The Unifier produces a Boolean result specifying whether or not the two automatons unify, and, if they do not unify, a list of the states and variables that do not unify. If the two automatons unify, they are considered verified.

For example, suppose Automaton1 and Automaton 2 have states representing declarations and executable statements. The states representing declarations in Automaton1 do not unify with states representing executable statements in Automaton2 so those pairs of states are discarded. Pairs of states that represent similar variable declarations or similar executable statements in the two automatons may unify.

6. Example

Figure 4 illustrates the INVERT verification process performed by the prototype tool. INVERT generates a trimmed automaton for both the first program, labeled Source1, and the second program, labeled Source2. Unnecessary code identified by the Trimmer is based on the list of desired outputs provided for each program. The two automatons are then subjected to the INVERT unification

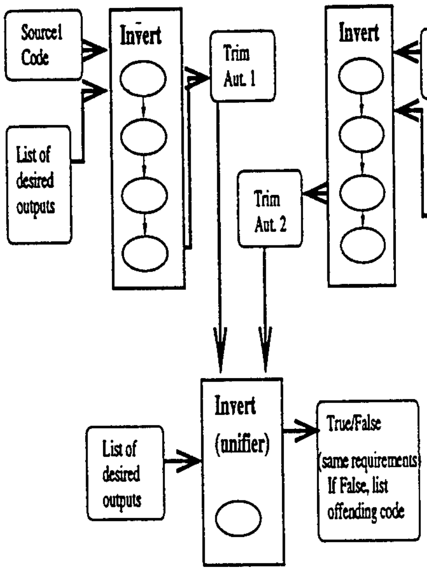

Figure 3: INVERT Verification Process.

process. Since the Unifier is still executed separately from the other components in the prototype INVERT, the list of desired outputs must again be specified. The Unifier reports a Boolean result as to whether or not the two programs unify, and, if they do not unify, the Unifier also reports the states and variables in each program that failed to unify.

Ultimately the translated-code version of INVERT will verify software translated from one language to another. However, as stated in Section 4, the prototype tool only accepts simple, procedural, non-real-time XPL programs with no input or output statements. Therefore the verification capability of the prototype INVERT was demonstrated by providing as input two different XPL programs that implement the same basic design using slightly different language constructs. Figure 5 lists a procedure named Bubblel and a procedure named Bubble2.

Both procedures sort an array of 10 numbers in ascending order using a bubble sort algorithm. However, Bubblel uses nested counted loops while Bubble2 uses nested while loops. In addition, Bubble2 has several lines of unnecessary and dead code, as indicated by the comments in the program.

INVERT is executed for both Bubblel and Bubble2 to generate trimmed automatons for each. The desired output for procedure Bubblel is the array A, and the desired output for procedure Bubble2 is array Arr. The Bubble2 automaton has the unnecessary and dead code flagged.

```
BUBBLE 1
**********************
* BUBBLE SORT ALGORITHM *
* USING COUNTED LOOPS *
* TO SORT 10 NUMBERS *
*DC = Dead Code *
***************************
DECLARE A(9) BIT(8)
   INITIAL(3,6,7,1,9,0,4,2,5,8)
DECLARE(TEMP, I, J) BIT(8)
DO I = 0 TO 8;

   DO J = I+1 TO 9
      IF A(J) < A(J)
      THEN DO;
         TEMP = A(J);
         A(J) = A(I);
         A(I) = TEMP;
      END;
   END;
END;
IF A(9) < A(0)
THEN TEMP = 0; * DC *
EOF
```

```
BUBBLE 2
*************************
* BUBBLE SORT ALGORITHM *
* WITH WHILE LOOPS *
* AND UNNECESSARY CODE 8
* UC = Unnecccessary Code *
*************************
DECLARE ARR(9) BIT(8)
   INITIAL(3,6,7,1,9,0,4,2,5,8)
DECLARE(T, T2, I, J)   BIT(8)
DECLARE LENGTH  BIT(8)
                 INITIAL(9)
J = 0;  *UC *
T2 = 0;        * UC 8
J = J+ 10;       * UC *
J = J - 10;       * UC *
DO WHILE J <= LENGTH - 1;
   I = J + 1;
   DO WHILE I <= LENGTH;
      IF ARR(I) < ARR(J)
      THEN DO;
         T = ARR(I);
      ARR(I) = ARR(J);
         ARR(J) = T;
         T2 =T;   * UC *
      END;
      I = I + 1;
      T2 = I;   * UC *
      I = ARR(O)+ARR(9);* UC *
      I = T2;   * UC *
   END;
   J = J + 1;
END;
IF ARR(9) < ARR(0)
THEN T = 0;      * DC *
EOF
```

Figure 4

After the trimmed automatons are generated for the two programs, the Unifier is executed. The two trimmed automatons, Bubblel and Bubble2, and the list of desired outputs for Bubblel (the array A) are provided as input. The Unifier performs variable substitution and determines that variable A in Bubblel and variable Arr in Bubble2 substitute because they are the same type, their values are the same in the first state and the last state, and they do not substitute with any other variables. The Unifier then performs unification on all the program states. It determines that the two programs unify because all of their states unify. Two states unify if the variables assigned in that state substitute with each other and are assigned the same value, and variables referenced in that state have the same value.

The two programs, Bubblel and Bubble2, are considered verified as performing the same functionally.

7. Future Work

Currently, the prototype tool can only statically verify software that uses the same language, same basic design, similar variables, and same set of inputs for desired outputs. Future work on the translated-code verification tool includes expanding the prototype to handle two full languages and adding the following capabilities: verify software with variable inputs as opposed to fixed inputs, verify different algorithms that perform the same function, and verify real-time code and hierarchical code. In addition, cosmetic enhancements such as an easy-to-use graphical user interface are needed to make the verification tool user friendly. A method of representing requirements and specifications in an automaton must be developed before the translated-code verification tool can be expanded into a code-to-requirements verification tool.

References

[1] Antsaklis, P., Kohn, W., Nerode, A, and Sastry, S. eds., *Hybrid Systems II*, Lecture Notes in Computer Science vol. 999, Springer-Verlag, (1995).

[2] Bowler, O., Grotzky, J., Nielson, M., Nilson,S., and Van Buren, J., "Requirements Analysis and Design Tools Report," Software Technology Support Center, Hill AFB, UT, April 1992.

[3] Boyer, R. S. and Moore, J.S., *A Computational Logic Handbook*, Academic Press, Corp.., San Diego, CA, 1988.

[4] Butler, Ricky W. "NASA Langley's Research Program in Formal Methods," Presented at the 6th Annual Conference on Computer Assurance (COMPASS '91), Gaithersburg, MD., June 24-28, 1991.

[5] Ge, X., Kohn, W., Nerode, A. and Remmel, J.B., "Feedback Derivations: Near Optimal Controls for Hybrid Systems", to appear in Hybrid Systems III, Springer Lecture Notes in Computer Science.

[6] Grossman, R.L., Nerode, A., Ravn, A. and Rischel, H. eds., *Hybrid Systems*, Lecture Notes in Computer Science 736, Springer-Verlag, (1993).

[7] Kohn, W. "A Formal Verification System For Functional Software Modules," Boeing Electronics report BE - 499- 08-86. August 1986.

[8] Kohn, W. and Nerode, A., "Foundations of Hybrid Systems" in Hybrid Systems, [6]

[9] Kohn W., and Nerode, A., "Multiple-Agent Hybrid Systems," Proc. IEEE CDC 1992, vol 4, pp 2956, 2972.

[10] Kohn W,, and Nerode A. "Models For Hybrid Systems: Automata, Topologies, Controllability, Observability" in Hybrid Systems, [6], 317-356.

[11] Kohn W. and Nerode, A., "Multiple Agent Hybrid Control Architecture" In *Logical Methods* (J. Crossley, J. B. Remmel, R. Shore, M. Sweedler, eds.), Birkhauser, (1993) 593-623.

[12] Kohn, W., Nerode, A. and Remmel, J.B., " Hybrid Systems as Finsler Manifolds: Finite State Control as Approximation to Connections", In [1], (1995)

[13] Kohn, W., Nerode, A. and Remmel, J.B., "Continualization: A Hybrid Systems Control Technique for Computing", Proceedings of CESA'96 IMACS Multiconference vol. 2, 5077-511.

[14] Kohn, W., Nerode, A. and Remmel, J.B., "Feedback Derivations: Near Optimal Controls for Hybrid Systems", Proceedings of CESA'96 IMACS Multiconference vol. 2, 517-521.

[15] Kohn, W., Nerode, A. and Remmel, J.B., "Scalable Data and Sensor Fusion via Multiple-Agent Hybrid Systems", submitted to IEEE Transactions of Automatic Control.

[16] Mills, H., "The Cleanroom Software Development Process," Crosstalk, No. 39, Software Technology Support Center, Hill AFB, UT, Dec. 1992.

[17] Price, G.,Daich, G.T., Murdock,D., and Hidden,E., "Test Preparation, Execution, and Analysis Tools Report," Software Technology Support Center, Hill AFB, UT, April 1992.

[18] Price, G., Ragland, B., Murdock, D., and Hidden,E., Source "Code Static Analysis Tools Report," Software Technology Support Center, Hill AFB, UT, April 1992.

[19] Rushby, J., "Quality Measures and Assurance for AI Software," Contractor Report 4187, NASA Langley Research Center, Hampton, VA., 1988.

[20] Rushby, J., von Henke,F., and Owre, S., "An Introduction to Formal Specification and Verification Using EHDM," Technical Report CSL-91-2, Computer Science Laboratory of SRI International, Menlo Park, CA, Feb. 1991.

[21] Wallace, D. R. and Fujii, R.U., "Software Verification and Validation: An Overview," IEEE Software, vol. 6 no. 3, pp. 10-17, May 1989.